#### **浅拷贝和深拷贝**

JavaScript有两种数据类型，基础数据类型和引用数据类型。基础数据类型都是按值访问的，我们可以直接操作保存在变量中的实际的值。而引用类型如Array，我们不能直接操作对象的堆内存空间。引用类型的值都是按引用访问的，即保存在变量对象中的一个地址，该地址与堆内存的实际值相关联。

****一、深拷贝和浅拷贝的区别****

  浅拷贝（shallow copy）：只复制指向某个对象的指针，而不复制对象本身，新旧对象共享一块内存；   
  深拷贝（deep copy）：复制并创建一个一摸一样的对象，不共享内存，修改新对象，旧对象保持不变。

var a = 25;var b = a;

b = 10;

console.log(a);//25

console.log(b);//10

//浅拷贝var obj1 = { a: 10, b: 20, c: 30 };var obj2 = obj1;

obj2.b = 40;

console.log(obj1);// { a: 10, b: 40, c: 30 }

console.log(obj2);// { a: 10, b: 40, c: 30 }

//深拷贝var obj1 = { a: 10, b: 20, c: 30 };var obj2 = { a: obj1.a, b: obj1.b, c: obj1.c };

obj2.b = 40;

console.log(obj1);// { a: 10, b: 20, c: 30 }

console.log(obj2);// { a: 10, b: 40, c: 30 }

****二、浅拷贝的实现****

var json1 = {"a":"name","arr1":[1,2,3]}function copy(obj1) {

　　　　var obj2 = {};

　　　　for (var i in obj1) {

　　　　　　obj2[i] = obj1[i];

　　　　}

　　　　return obj2;

}var json2 = copy(json1);

json1.arr1.push(4);

alert(json1.arr1); //1234

alert(json2.arr1) //1234

****三、深拷贝的实现****

****1、**Object.assign()**

let foo = {

a: 1,

b: 2,

c: {

d: 1,

}

}

let bar = {};

Object.assign(bar, foo);

foo.a++;

foo.a === 2 //true

bar.a === 1 //true

foo.c.d++;

foo.c.d === 2 //true

bar.c.d === 1 //false

bar.c.d === 2 //true

  Object.assign()是一种可以对非嵌套对象进行深拷贝的方法，如果对象中出现嵌套情况，那么其对被嵌套对象的行为就成了普通的浅拷贝。

****2、转成JSON****

  用JSON.stringify把对象转成字符串，再用JSON.parse把字符串转成新的对象。

var obj1 = { body: { a: 10 } };

var obj2 = JSON.parse(JSON.stringify(obj1));

obj2.body.a = 20;

console.log(obj1); // { body: { a: 10 } }

console.log(obj2); // { body: { a: 20 } }

console.log(obj1 === obj2); // false

console.log(obj1.body === obj2.body); // false

  但这种方法的缺陷是会破坏原型链，并且无法拷贝属性值为function的属性

****3、递归****

  采用递归的方法去复制拷贝对象

var json1={"name":"shauna","age":18,"arr1":[1,2,3,4,5],"string":'got7',"arr2":[1,2,3,4,5],"arr3":[{"name1":"shauna"},{"job":"web"}]};var json2={};function copy(obj1,obj2){

var obj2=obj2||{};

for(var name in obj1){

if(typeof obj1[name] === "object"){

obj2[name]= (obj1[name].constructor===Array)?[]:{};

copy(obj1[name],obj2[name]);

}else{

obj2[name]=obj1[name];

}

}

return obj2;

}

json2=copy(json1,json2)

json1.arr1.push(6);

alert(json1.arr1); //123456

alert(json2.arr1); //12345

#### **new 操作符**

一. new是干嘛的?

new操作符用来生成一个新的对象, 它后面必须跟上一个函数(否则, 会抛出TypeError异常), 这个

函数就是我们常说的构造函数.

二. new操作构造函数生成实例的过程

(1) 首先, 当我们使用new操作符时, js会先创建一个空的对象;

(2) 然后, 构造函数中的this指向该空对象;

(3) 其次, 在构造函数中通过操作this, 来给这个空对象赋予相应的属性;

(4) 最后, 返回这个经过处理的"空对象"(此时, 对象已经不是空的了).

****三.****new操作构造函数的注意事项

(1) 如果构造函数的返回值是一个原始类型(非引用对象, 如字符串), 那么返回值为new创建的"空对象", 如:

       var obj = new function() { return "code";};  
       alert(obj); // [object object]

(2) 如果构造函数的返回值是一个引用对象(数组, 对象, 函数等), 那么返回值会覆盖new创建的"空对象", 如:  
       var obj = new function() { return new String("code"); };  
       alert(obj); // "code"

<https://www.cnblogs.com/onepixel/p/5043523.html>

#### **原型是什么，原型的主要作用**



分析一下，上面这个例子！我们可以知道 People的类型是一个对象！  
按照javascript的说法，function定义的这个Person就是一个Object(对象),而且还是一个很特殊的对象，这个使用function定义的对象与使用new操作符生成的对象之间有一个重要的区别。这个区别就是function定义的对象有一个prototype属性，使用new生成的对象就没有这个prototype属性，我们一般称为普通对象！  
我们需要理解记忆以下的逻辑顺序：  
Person是一个对象，它有一个prototype的原型属性（因为所有的对象都一prototype原型！）prototype属性有自己的prototype对象，而pototype对象肯定也有自己的constuct属性，construct属性有自己的constuctor对象，神奇的事情要发生了，这最后一个constructor对象就是我们构造出来的function函数本身！

二、这样绕人的逻辑，总想一探究竟，要记得这是一种规定的逻辑思维，我们要做的是慢慢接受这种逻辑思维！我们不理论研究，用实例说话：



三、下面我们不免有了疑问，学这么绕的东西干什么，它给我们最实际的用处就是我们可以用原型来创建对象的属性和方法！我们不用它不也是可以创建属性和方法！这里是有区别的，既然不一样就有存在的价值！  
     我们可以通过给原型添加属性和方法来给给对象添加属性或方法！  
    Hero.prototype.name;  
    Hero.prototype.sayMe = function(){"添加对象的方法其实就是添加函数"}  
  
    让我们再深一步：当我们给对象添加了同名的属性或方法时会发生什么？



我们验证了这个例子得到得到了一些结论：  
当函数对象本身的属性或方法与原型的属性或方法同名的时候：  
    1、默认调用的是函数对象本身的属性或方法.  
    2、通过原型增加的属性或方法的确是存在的.  
    3、函数对象本身的属性或方法的优先级要高于原型的属性或方法.

#### **清除浮动的几种方法**

如何脱离标准文档流，css中给了三种方法，使一个元素脱离标准文档流。

   1.浮动。   2.绝对定位。  3.固定定位。

一、 什么是浮动呢？定义这样说：浮动的框可以向左或向右移动，直到它的外边缘碰到**包含框**或**另一个浮动框的边框**为止。 其实也就是把快级元素变成能和别的快级并排成一行。

二、浮动的的元素有4点特性：

    1.浮动元素会脱离标准文档流，已经不区分快和行了。

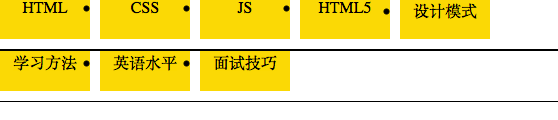
    2.浮动的元素会互相贴靠。

    3.浮动的元素有“字围”的效果。

    4.收缩。一个浮动的元素，如果没有设置width，那么将自动收缩为文字的宽度。

三、清除浮动的方法。

   1.给浮动的元素的祖先元素加高度（父元素或祖元素）。只要浮动在一个有高度的盒子中，那么这个浮动就不会影响后面的元素了，也就是说，如果盒子内部有浮动，这个盒子有高，那么妥妥的，浮动不会互相影响。所以也就清除了浮动带来的影响。浮动的元素，只能被有高度的盒子关住。但是，工作上，我们绝对不会给所有的盒子加高度，这是因为麻烦，并且不能适应页面的快速变化。（不建议使用）

    .

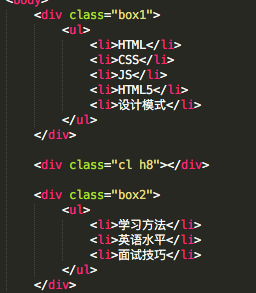
   2.clear：both；清除掉两边的浮动。但是有一个非常致命的问题，margin失效了，上下之间没有了间距。



3.隔墙法。就是在两个部分之间加一个<div>，设置一个高度。还有就是“内墙发”：就是在块里边在放一个<div>，设置高度，不仅不让后边的去追前边的，并且能给外边的<div>撑出一个高度。

  注：脱标的元素是不能把父元素撑出高度滴。

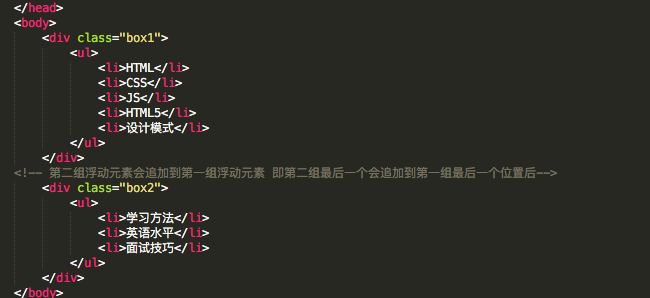
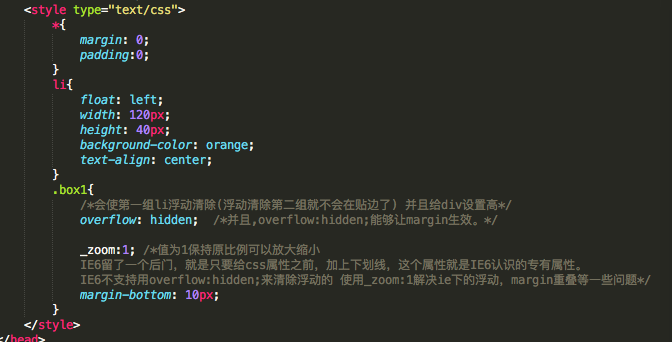
                外墙法：

       内墙法：



   4.overflow：hidden；  一个父元素不能被自己浮动的儿子撑出高度。但是，只要给父亲加上 overflow：hidden；那么父元素就能被儿子撑出高度了。  最最最重要的是，使用overflow：hidden；能使margin生效！你没有看错，就是酱。

#### **h5和css3的新特性**

#### **前端优化方法**

#### **兼容性有几点体现**

#### **判断对象的数据类型的几种方法**

#### **JS复杂数据类型**

#### **微信小程序、微信浏览器等**

#### **HTTP响应码**

<https://blog.csdn.net/zmx729618/article/details/51576489>

#### **版本管理工具**

#### **打包工具**

<https://www.cnblogs.com/chuaWeb/p/gulp.html>

#### **Call和apply的区别**

<http://www.cnblogs.com/onepixel/p/5143863.html>

在JavaScript 中，call、apply 和 bind 是 Function 对象自带的三个方法，这三个方法的主要作用是改变函数中的 this 指向，从而可以达到`接花移木`的效果。本文将对这三个方法进行详细的讲解，并列出几个经典应用场景。

#### ****1、call**(thisArgs [,args...])**

该方法可以传递一个thisArgs参数和一个参数列表，thisArgs 指定了函数在运行期的调用者，也就是函数中的 this 对象，而参数列表会被传入调用函数中。thisArgs 的取值有以下四种情况：

* 不传，或者传null,undefined， 函数中的 this 指向 window 对象
* 传递另一个函数的函数名，函数中的 this 指向这个函数的引用
* 传递字符串、数值或布尔类型等基础类型，函数中的 this 指向其对应的包装对象，如 String、Number、Boolean
* 传递一个对象，函数中的 this 指向这个对象

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | function a(){      console.log(this);//输出函数a中的this对象  }  function b(){}//定义函数b    var obj = {name:'onepixel'};//定义对象obj    a.call();//window  a.call(null);//window  a.call(undefined);//window  a.call(1);//Number  a.call('');//String  a.call(true);//Boolean  a.call(b);// function b(){}  a.call(obj);//Object |

这是call 的核心功能，它允许你在一个对象上调用该对象没有定义的方法，并且这个方法可以访问该对象中的属性，至于这样做有什么好处，我待会再讲，我们先看一个简单的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var a = {        name:'onepixel',//定义a的属性        say:function(){//定义a的方法          console.log("Hi,I'm function a!");      }  };    function b(name){      console.log("Post params: "+ name);      console.log("I'm "+this.name);      this.say();  }    b.call(a,'test');  >>  Postparams: test  I'm onepixel  I'm function a! |

当执行b.call 时，字符串`test`作为参数传递给了函数b,由于call的作用，函数b中的this指向了对象a, 因此相当于调用了对象a上的函数b,而实际上a中没有定义b 。

#### ****2、apply**(thisArgs [,args[]])**

apply 和 call 的唯一区别是第二个参数的传递方式不同，apply 的第二个参数必须是一个数组，而 call 允许传递一个参数列表。值得你注意的是，虽然 apply 接收的是一个参数数组，但在传递给调用函数时，却是以参数列表的形式传递，我们看个简单的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5 | function b(x,y,z){      console.log(x,y,z);  }    b.apply(null,[1,2,3]);// 1 2 3 |

apply 的这个特性很重要，我们会在下面的应用场景中提到这个特性。

#### ****3、bind**(thisArgs [,args...])**

bind是ES5 新增的一个方法，它的传参和call类似，但又和 call/apply 有着显著的不同，即调用 call 或 apply 都会自动执行对应的函数，而 bind 不会执行对应的函数，只是返回了对函数的引用。粗略一看，bind 似乎比call/apply 要落后一些，那ES5为什么还要引入bind 呢？

其实，ES5引入 bind 的真正目的是为了弥补 call/apply 的不足，由于 call/apply 会对目标函数自动执行，从而导致它无法在事件绑定函数中使用，因为事件绑定函数不需要我们手动执行，它是在事件被触发时由JS 内部自动执行的。而 bind 在实现改变函数 this 的同时又不会自动执行目标函数，因此可以完美的解决上述问题，看一个例子就能明白：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var obj = {name:'onepixel'};    /\*\*   \* 给document添加click事件监听，并绑定onClick函数   \* 通过bind方法设置onClick的this为obj，并传递参数p1,p2   \*/  document.addEventListener('click',onClick.bind(obj,'p1','p2'),false);    //当点击网页时触发并执行  function onClick(a,b){      console.log(              this.name,//onepixel              a,//p1              b //p2      )  } |

当点击网页时，onClick 被触发执行，输出onepixel p1 p2, 说明 onClick 中的 this 被 bind 改变成了obj 对象，为了对 bind 进行深入的理解，我们来看一下 bind 的 polyfill 实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | if (!Function.prototype.bind) {      Function.prototype.bind = function (oThis) {          var aArgs = Array.prototype.slice.call(arguments, 1),              fToBind =this,// this在这里指向的是目标函数              fBound = function () {                  return fToBind.apply(                      // 如果外部执行var obj = new fBound(),则将obj作为最终的this，放弃使用oThis                      this instanceof fToBind                              ?this  // 此时的this就是new出的obj                              : oThis ||this,// 如果传递的oThis无效，就将fBound的调用者作为this                        // 将通过bind传递的参数和调用时传递的参数进行合并，并作为最终的参数传递                      aArgs.concat(Array.prototype.slice.call(arguments)));              };            // 将目标函数的原型对象拷贝到新函数中，因为目标函数有可能被当作构造函数使用          fBound.prototype =this.prototype;            // 返回fBond的引用，由外部按需调用          return fBound;      };  } |

#### **一旦函数通过bind传递了有效的this对象，则该函数在运行期的this将指向这个对象，即使通过call或apply来试图改变this的指向也是徒劳的。**

// 实现数组的去重功能

Array.prototype.unique = function(fn) {

        var rst = [];

        var tmp = {};

        this.forEach(function(val) {

            // 使用call来改变fn的this指向，这里传window

            var key ='uniq' + (typeof fn ==='function' ? fn.call(window, val) : val);

            if (!tmp.hasOwnProperty(key)) {

                rst.push(val);

                tmp[key] =null;

            }

        },this);

        return rst;

}

// 对象数组去重

var arr = [

    { id: 2 }, { id: 4 }, { id: 3 }, { id: 3 }, { id: 4 }, { id: 6 }

]

arr.unique(function(v) {

     console.log(this)// 使用bind传递了Array,则this一定是Array,而不会是window

     return v.id

}.bind(Array));

#### ****4、应用场景一：继承****

大家知道，JavaScript中没有诸如Java、C# 等高级语言中的extend 关键字，因此JS 中没有继承的概念，如果一定要继承的话，call 和 apply 可以实现这个功能：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | function Animal(name,weight){     this.name = name;     this.weight = weight;  }    function Cat(){      Animal.call(this,'cat','50');    //Animal.apply(this,['cat','50']);       this.say = function(){        console.log("I am " +this.name+",my weight is " +this.weight);     }  }    var cat =new Cat();  cat.say();//I am cat,my weight is 50 |

当通过new 运算符产生了cat 时，Cat中的 this 就指向了cat对象(关于new运算符的讲解，请参考[JS构造函数和new运算符](http://www.cnblogs.com/onepixel/p/5043523.html" \t "http://www.cnblogs.com/onepixel/p/_blank)，而继承的关键是在于Cat中执行了Animal.call(this,'cat','50') 这句话，在call中将this作为thisArgs参数传递，于是Animal 方法中的 this 就指向了Cat中的 this，而 cat 中的 this 指向的是 cat 对象，所以Animal 中的 this 指向的就是 cat 对象，在 Animal 中定义了name 和 weight 属性，就相当于在 cat 中定义了这些属性，因此 cat 对象便拥有了Animal 中定义的属性，从而达到了继承的目的。

#### ****5、应用场景二：移花接木****

在讲下面的内容之前，我们首先来认识一下JavaScript 中的一个非标准专业术语：**ArrayLike**(类数组/伪数组)

ArrayLike 对象即拥有数组的一部分行为，在DOM 中早已表现出来，而jQuery 的崛起让ArrayLike 在JavaScript 中大放异彩。ArrayLike 对象的精妙在于它和JS 原生的 Array 类似，但是它是自由构建的，它来自开发者对JavaScript 对象的扩展，也就是说：对于它的原型(prototype)我们可以自由定义，而不会污染到JS原生的Array。

ArrayLike 对象在JS中被广泛使用，比如DOM 中的NodeList, 函数中的arguments 都是类数组对象，这些对象像数组一样存储着每一个元素，但它没有操作数组的方法，而我们可以通过call 将数组的某些方法`移接`到ArrayLike 对象，从而达到操作其元素的目的。比如我们可以这样遍历函数中的arguments:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | function test(){      // 检测arguments是否为Array的实例      console.log(              arguments instanceof Array,//false              Array.isArray(arguments) //false      );      // 判断arguments是否有forEach方法      console.log(arguments.forEach);//undefined        // 将数组中的forEach应用到arguments上      Array.prototype.forEach.call(arguments,function(item){          console.log(item);// 1 2 3 4      });    }  test(1,2,3,4); |

除此之外，对于apply 而言，我们上面提到了它独有的一个特性，即apply 接收的是数组，在传递给调用函数的时候是以参数列表传递的。 这个特性让apply 看起来比call 略胜一筹，比如有这样一个场景：给定一个数组[1,3,4,7]，然后求数组中的最大元素，而我们知道，数组中并没有获取最大值的方法，一般情况下，你需要通过编写代码来实现。而我们知道，Math 对象中有一个获取最大值的方法，即Math.max()， max方法需要传递一个参数列表，然后返回这些参数中的最大值。而apply 不仅可以将Math 对象的max 方法应用到其他对象上，还可以将一个数组转化为参数列表传递给max,看代码就能一目了然：

|  |  |
| --- | --- |
| 1  2  3 | var arr = [2,3,1,5,4];    Math.max.apply(null,arr);// 5 |